A Fast Implementation of the ISODATA Clustering Algorithm[*]

Nargess Memarsadeghi[†] David M. Mount[‡] Nathan S. Netanyahu[§]
Jacqueline Le Moigne[¶]

Submitted to IJCGA
December 31, 2005

Abstract

Clustering is central to many image processing and remote sensing applications. ISODATA is one of the most popular and widely used clustering methods in geoscience applications, but it can run slowly, particularly with large data sets. We present a more efficient approach to ISODATA clustering, which achieves better running times by storing the points in a kd-tree and through a modification of the way in which the algorithm estimates the dispersion of each cluster. We also present an approximate version of the algorithm which allows the user to further improve the running time, at the expense of lower fidelity in computing the nearest cluster center to each point. We provide both theoretical and empirical justification that our modified approach produces clusterings that are very similar to those produced by the standard ISODATA approach. We also provide empirical studies on both synthetic data and remotely sensed Landsat and MODIS images that show that our approach has significantly lower running times.

[*]A preliminary version of this paper appeared in *Proceedings of the IEEE International Geoscience and Remote Sensing Symposium* (IGARSS'03), Toulouse, France, 2003, Vol. III, 2057–2059.

[†]NASA Goddard Space Flight Center, Architecture and Automation Branch, Greenbelt, MD 20771 and Department of Computer Science, University of Maryland, College Park, Maryland, 20742. Email: nargess@cs.umd.edu.

[‡]Department of Computer Science, University of Maryland, College Park, Maryland, 20742. The work of this author was supported by the Science Foundation under grant CCR-0098151. Email: mount@cs.umd.edu.

[§]Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel, and Center for Automation Research, University of Maryland, College Park, Maryland, 20742. Email: nathan@cs.biu.ac.il.

[¶]NASA Goddard Space Flight Center, previously Applied Information Sciences Branch, currently Advanced Architectures and Automation Branch, Greenbelt, MD 20771. Email: Jacqueline.LeMoigne@nasa.gov.

1 Introduction

Unsupervised clustering is a fundamental tool in image processing for geoscience and remote sensing applications. For example, unsupervised clustering is often used to obtain vegetation maps of an area of interest. This approach is useful when reliable training data are either scarce or expensive, and when relatively little *a priori* information about the data is available. Unsupervised clustering methods play a significant role in the pursuit of unsupervised classification [39].

The problem of clustering points in multidimensional space can be posed formally as one of a number of well-known optimization problems, such as the Euclidean *k-median problem* [22], in which the objective is to minimize the sum of distances to the nearest center, the Euclidean *k-center problem* [16], in which the objective is to minimize the maximum distance, and the *k-means problem*, in which the objective is to minimize the sum of squared distances [15, 23, 32, 33]. Efficient solutions are known to exist only in special cases, e.g., the planar 2-center problem [1, 41]. There are no efficient exact solutions known to any of these problems for general k, and some formulations are known to be NP-hard [14]. Efficient approximation algorithms have been developed in some cases. These include constant factor approximations for the k-center problem [9, 16], the k-median problem [24, 8, 4], and the k-means problem [26]. There are also ϵ-approximation algorithms for the k-median [2, 30] and k-means [35, 31] problems, including improvements based on coresets [20, 19]. Work on the k-center algorithm for moving data points, as well as a linear time implementation of a 2-factor approximation of the k-center problem have also been introduced [17, 18].

In spite of progress on theoretical bounds, ϵ-approximation algorithms for these clustering problems are still not suitable for practical implementation in multidimensional spaces, when k is not a small constant. This is due to very fast growing dependencies in the asymptotic running times on the dimension and/or on k. In practice, it is common to use heuristic approaches, which seek to find a reasonably good clustering, but do not provide guarantees on the quality of the results. This includes randomized approaches, such as CLARA [28] and CLARANS [36], and methods based on neural networks [29]. One of the most popular and widely used clustering heuristics in remote sensing is ISODATA [5, 23, 25, 42]. A set of n data points in d-dimensional space is given along with an integer k indicating the initial number of clusters and a number of additional parameters. The general goal is to compute a set of cluster centers in d-space. Although there is no specific optimization criterion, the algorithm is similar in spirit to the well-known k-means clustering method [23], in which the objective is to minimize the average squared distance of each point to its nearest center, called the *average distortion*. One significant advantage of ISODATA over k-means is that the user need only provide an initial estimate of the number of clusters, and based on various heuristics the algorithm may alter the number of clusters by either deleting small clusters, merging nearby clusters, or splitting large diffuse clusters. The algorithm will be described in the next section.

As currently implemented, ISODATA can run very slowly, particularly on large data sets. Given its wide use in remote sensing, its efficient computation is an important goal. Our objective in this paper is not to provide a new or better clustering algorithm, but rather, to show how computational geometry methods can be applied to produce a faster implementation of ISODATA clustering. There are a number of minor variations of ISODATA that appear in the literature. These variations involve issues such as termination conditions, but they are equivalent in terms of their overall structure. We focus on a widely used version, called ISOCLUS [37], which will be presented in the next section.

The running times of ISODATA and ISOCLUS are dominated by the time needed to compute the nearest among the k cluster centers to each of the n points. This can be reduced to the problem of answering n nearest-neighbor queries over a set of size k, which naively would involve $O(kn)$ time. To improve the running time, an obvious alternative would be to store the k centers in a spatial index, e.g., a kd-tree [6]. However, this is not the best approach, because k is typically much smaller than n, and the center points are constantly changing, requiring the tree to be constantly updated. Kanungo *et al.* [27] proposed a more efficient and practical approach by storing the points, rather than the cluster centers, in a kd-tree. The tree is then used to solve the *reverse nearest neighbor problem*, that is, for each center we compute the set of points for which this center is the closest. This method is called the *filtering algorithm*.

We show how to modify this approach for ISOCLUS. The modifications are not trivial. First off, in order to perform the sort of aggregate processing that the filtering algorithm employs, it was necessary to modify the way in which the ISOCLUS algorithm computes the degree of dispersion within each cluster. In Section 5 and Section 6 we present, respectively, empirical and theoretical justification that this modification does not significantly alter the nature of the clusters that the algorithm produces. In order to further improve execution times, we have also introduced an approximate version of the filtering algorithm. A user-supplied approximation error bound $\epsilon > 0$ is provided to the algorithm, and each point is associated with a center whose distance from the point is not farther than $(1 + \epsilon)$ times the distance to its true nearest neighbor. This result may be of independent interest because it can be applied to k-means clustering as well. It is presented in Section 3.5.

The running time of the filtering algorithm is a subtle function of the structure of the clusters and centers, and so rather than presenting a worst-case asymptotic analysis, we present an empirical analysis of its efficiency based on both synthetically generated data sets, and actual data sets from a common application in remote sensing and geostatistics. These results are presented in Section 5. As the experiments show, depending on the various input parameters (that is, dimension, data size, number of centers, etc.), the algorithm presented runs faster than a straightforward implementation of ISOCLUS, by factors ranging from 1.3 to over 50. In particular, the improvements are very good for typical applications in geostatistics, where the data size n and the number of centers k are large, and the dimension d is relatively small. Thus, we feel that this algorithm can play an important role in the analysis of geostatistical data analysis and other applications of data clustering.

The remainder of the paper is organized as follows. We start in the next section with a description of ISOCLUS, which is a variant of ISODATA that we have focused on. In Section 3 we provide background, concerning basic tools such as the kd-tree data structure and the filtering algorithm, that will be needed in our efficient implementation of ISOCLUS. We present, in Section 4, our improved variants of the ISOCLUS algorithm, and in Section 5 the experimental results for these variants. In Section 6 we provide a theoretical justification of our cluster dispersion measure, which formed the basis of our efficient implementation. Finally, Section 7 contains concluding remarks.

2 The ISOCLUS Algorithm

We begin by presenting the particular variant of ISODATA, called ISOCLUS [37], that will be modified. Although our description is not exhaustive, it contains enough information to understand our various modifications. The algorithm tries to find the best cluster centers through an iterative approach. It also uses a number of different heuristics to determine whether to merge or split clusters.

At a high level, the following tasks are performed in each iteration of the algorithm: Points are assigned to their closest cluster centers, cluster centers are updated to be the centroid of their associated points, clusters with very few points are deleted, large clusters satisfying some conditions are split, and small clusters satisfying other conditions are merged. The algorithm continues until the number of iterations exceeds a user-supplied value.

Let us present the algorithm in more detail. There are a number of user-supplied parameters. These include the following. (In parentheses we give the variable name of the parameter used in [37].)

k_{init}: initial number of clusters (NUMCLUS)
$n_{\min}$: minimum number of points that can form a cluster (SAMPRM)
$I_{\max}$: maximum number of iterations (MAXITER)
$\sigma_{\max}$: maximum standard deviation of points from their cluster center along each axis (STDV)
$L_{\min}$: minimum required distance between two cluster centers (LUMP)
$P_{\max}$: maximum number of cluster pairs that can be merged per iteration (MAXPAIR)

Here is an overview of the algorithm. (See [37] for details.) Let $S = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ denote the set of points to be clustered. Each point $\mathbf{x}_j = (x_{j1}, \ldots, x_{jd})$ is treated as a vector in real d-dimensional space, $\mathbb{R}^d$. Let n denote the number of points. If the original set is too large, all of the iterations of the algorithm, except the last, can be performed on a random subset of S of an appropriate size. Throughout, let $\|\mathbf{x}\|$ denote the Euclidean length of the vector $\mathbf{x}$.

(1) Letting $k = k_{\text{init}}$, randomly sample k cluster initial centers $Z = \{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_k\}$ from S.

(2) Assign each point to its closest cluster center. For $1 \le i \le k$, let $S_i \subseteq S$ be the subset of points that are closer to $\mathbf{z}_i$ than to any other cluster center of Z. That is, for any $\mathbf{x} \in S$,

$$\mathbf{x} \in S_j \quad \text{if} \quad \|\mathbf{x} - \mathbf{z}_j\| < \|\mathbf{x} - \mathbf{z}_i\|, \quad \forall i \neq j.$$

(Ties for the closest center are broken arbitrarily.) Let n_j denote the number of points of S_j.

(3) Remove cluster centers with fewer than $n_{\min}$ points. (The associated points of S are not deleted, but are ignored for the remainder of the iteration.) Adjust the value of k and relabel the remaining clusters $S_1 \ldots, S_k$ accordingly.

(4) Move each cluster center to the centroid of the associated set of points. That is,

$$\mathbf{z}_j \leftarrow \frac{1}{n_j} \sum_{\mathbf{x} \in S_j} \mathbf{x}, \qquad \text{for } 1 \le j \le k.$$

If any clusters were deleted in Step 3, then the algorithm goes back to Step 2.

(5) Let Δ_j be the average distance of points of S_j to the associated cluster center $\mathbf{z}_j$, and let Δ be the overall average of these distances.

$$\Delta_j \leftarrow \frac{1}{n_j} \sum_{\mathbf{x} \in S_j} \|\mathbf{x} - \mathbf{z}_j\|, \quad \text{for } 1 \le j \le k. \qquad \Delta \leftarrow \frac{1}{n} \sum_{j=1}^{k} n_j \Delta_j.$$

(6) If this is the last iteration, then set $L_{\text{min}} = 0$ and go to Step 9. Also, if $2k > k_{\text{init}}$ and it is either an even numbered iteration or $k \geq 2k_{\text{init}}$, then go to Step 9.

(7) For each cluster S_j, compute a vector $\mathbf{v}_j = (v_1, \ldots, v_d)$ whose ith coordinate is the standard deviation of the ith coordinates of the vectors directed from $\mathbf{z}_j$ to every point of S_j. That is,

$$v_{ji} \leftarrow \left(\frac{1}{n_j} \sum_{\mathbf{x} \in S_j} (x_i - z_{ji})^2 \right)^{1/2} \qquad \text{for } 1 \leq j \leq k \text{ and } 1 \leq i \leq d.$$

Let $v_{j,\text{max}}$ denote the largest coordinate of $\mathbf{v}_j$.

(8) For each cluster S_j, if $v_{j,\text{max}} > \sigma_{\text{max}}$ and either

$$((\Delta_j > \Delta) \ \text{ and } \ (n_j > 2(n_{\text{min}} + 1))) \ \text{ or } \ k \leq \frac{k_{\text{init}}}{2},$$

then increment k and split S_j into two clusters by replacing its center with two cluster centers centered around $\mathbf{z}_j$ and separated by an amount and direction that depends on $v_{j,\text{max}}$ [37]. If any clusters are split in this step, then go to Step 2.

(9) Compute the pairwise *intercluster distances* between all distinct pairs of cluster centers

$$d_{ij} \leftarrow \|\mathbf{z}_i - \mathbf{z}_j\|, \qquad \text{for } 1 \leq i < j \leq k.$$

(10) Sort the intercluster distances of Step 9 in increasing order, and select a subset of at most P_{max} of the closest such pairs of clusters, such that each pair has an intercluster distance of at most L_{min}. For each such pair (i, j), if neither S_i nor S_j has been involved in a merger in this iteration, replace the two clusters S_i and S_j with a merged cluster $S_i \cup S_j$, whose associated cluster center is their weighted average

$$\mathbf{z}_{ij} \leftarrow \frac{1}{n_i + n_j}(n_i \mathbf{z}_i + n_j \mathbf{z}_j).$$

Relabel the remaining clusters and decrease k accordingly.

(11) If the number of iterations is less than I_{max}, then go to Step 2.

If the algorithm is implemented in the most straightforward manner, and if it is assumed that the number of clusters, k, is much smaller than the total number of points, n, then the most time-consuming stage of the algorithm is Step 2. Computing naively the distances from each of the n points of S to each of the k centers for a total of $O(kn)$ time (assuming a fixed dimension d).

Our approach for improving the algorithm's running time is to speed up Step 2 through the use of an appropriate spatial data structure. Note that the algorithm does not need to explicitly compute the closest center to each point. What is needed is the centroid of the points that are closest to each center. Our approach is to compute this quantity directly. Before describing how to do this, we provide some background on a related clustering algorithm, called *Lloyd's algorithm*, and its fast implementation by a method called the *filtering algorithm*.

3 The Filtering Algorithm

At its heart, the ISOCLUS algorithm is based on an enhancement of a simple and widely used heuristic for k-means clustering, sometimes called *Lloyd's algorithm* or the *k-means algorithm* [11, 32, 33]. It iteratively repeats the following two steps until convergence. First, for each cluster center, it computes the set of points for which this center is the closest. Next, it moves each center to the centroid of its associated set. It can be shown that with each step the average distortion decreases and that the algorithm converges to a local minimum [40]. See [21, 7, 34, 38] for further discussion on the statistical properties and convergence conditions of Lloyd's algorithm and other related procedures. The ISOCLUS algorithm combines Lloyd's algorithm with additional mechanisms for eliminating very small clusters (Step 3), splitting large clusters (Steps 7–8), and merging nearby clusters (Steps 9–10).

As with ISOCLUS, the running time of Lloyd's algorithm is dominated by the time to compute the nearest cluster center to each data point. Naively, this would require $O(kn)$ time. Kanungo *et al.* [27] presented a more efficient implementation of Lloyd's algorithm, called the *filtering algorithm*. Although its worst-case asymptotic running time is not better than the naive algorithm, this approach was shown to be quite efficient in practice. In this section we present a high-level description of the filtering algorithm. We also introduce an approximate version of this algorithm, in which points may be assigned, not to their nearest neighbor, but to an approximate nearest neighbor.

3.1 The kd-tree

If considered at a high level, the filtering process implicitly involves computing, for each of the k centers, some aggregate information for all the points that are closer to this center than any other. In particular, it needs to compute the centroid of these points and some other statistical information that is used by the ISOCLUS algorithm. Thus, the process can be viewed very abstractly as answering a number of range queries involving k disjoint ranges, each being the Voronoi cell of some cluster center. As such, an approach based on hierarchical spatial subdivisions is natural.

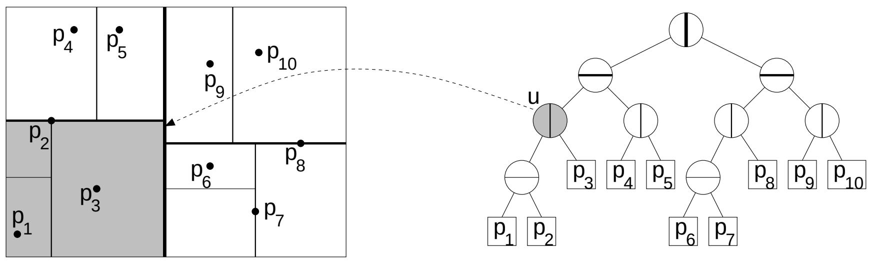


Fig. 1: An example of a kd-tree of a set of points in the plane, showing both the associated spatial subdivision (left) and the binary tree structure (right).

The filtering algorithm builds a standard kd-tree [6], augmented with additional statistical information, which will be discussed below. A *kd-tree* is a hierarchical decomposition of space axis-aligned hyperrectangles called *cells*. Each node of the tree is implicitly associated with a unique cell and the subset of the points that lie within this cell. Each *internal node* of the kd-tree stores an axis-orthogonal *splitting hyperplane*. This hyperplane subdivides the cell into two subcells, which are associated with the left and right subtrees of the node. Nodes holding a single point are declared to be *leaves* of the tree. In Fig. 1, the highlighted node u of the tree is associated with the shaded rectangular cell shown on the left side of the figure and the subset $\{p_1, p_2, p_3\}$ of points. It is well known that a kd-tree on n points can be constructed in $O(n \log n)$ time [12].

3.2 The Filtering Process

We provide an overview of how the filtering algorithm is used to perform one iteration of Lloyd's algorithm. (See [27] for details.) Given a kd-tree for the data points S and the current set of k center points, the algorithm processes the nodes of the kd-tree in a top-down recursive manner, starting at the root. Consider some node u of the tree. Let $S(u)$ denote the subset of points S that are associated with this node. If it can be inferred that all the points of $S(u)$ are closer to some center $\mathbf{z}_j$ than to any other center (that is, the node's associated rectangular cell lies entirely within the Voronoi cell of $\mathbf{z}_j$), then we may *assign* u to cluster S_j. Every point associated with u is thus *implicitly assigned* to this cluster. (For example, this is the case for the node associated with cell a shown in Fig. 2.) If this cannot be inferred, then the cell is split, and we apply the process recursively to its two children. (This is the case for the node associated with cell b in the figure, which is split and whose two children are b_1 and b_2.) Finally, if the process arrives at a leaf node, which contains a single point, then we determine which center is closest to the point, and assign its associated node to this center. (This is the case for the node associated with cell c of the figure.)

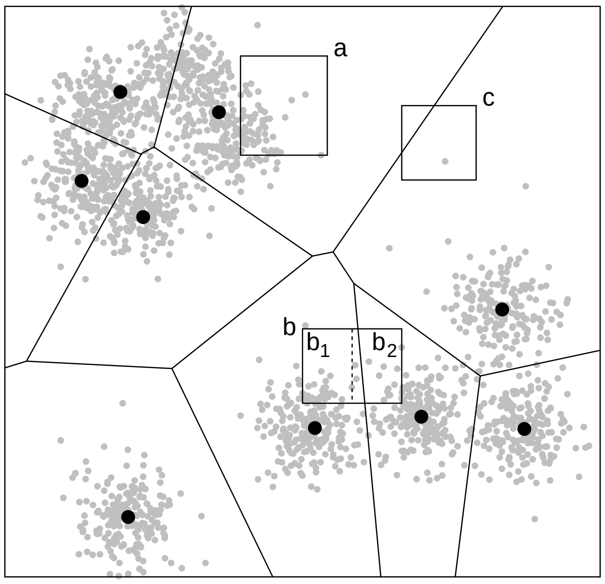


Fig. 2: Classifying nodes in the filtering algorithm. The subdivision is the Voronoi diagram of the centers, which indicates the neighborhood regions of each center.

At the conclusion of the process, the filtering algorithm assigns the nodes of the kd-tree to clusters in such a manner that every point of S is implicitly assigned to its closest cluster center. Furthermore, this is done so that the sets $S(u)$ assigned to a given cluster form a disjoint union of the associated cluster. There are two issues to be considered: (1) How to determine whether one center is closer to every point of a node's cell than all other centers, and (2) when this occurs, how to assign *en masse* the points of the node to this center. We address these issues in reverse order in the next two sections.

3.3 Additional Statistical Information

As mentioned above, the k-means algorithm seeks a placement of the centers that minimizes the average squared distance of each point to its nearest center. More formally, for each cluster S_j, we recall that $n_j = |S_j|$, and define the *average distortion* of the jth cluster, denoted $\Delta_j^{(2)}$, to be the average squared distance of each point in cluster S_j to its cluster center, that is,

$$\Delta_j^{(2)} = \frac{1}{n_j} \sum_{\mathbf{x} \in S_j} \|\mathbf{x} - \mathbf{z}_j\|^2.$$

(Contrast this quantity with the average distance Δ_j, computed in Step 5 of the ISOCLUS algorithm.) The overall distortion of the entire data set is the weighted average distortion among all clusters, where the weight factor for the jth cluster is n_j/n, that is, the fraction of points in this cluster.

In order to compute this information efficiently for each cluster, we store the following statistical information with each node u of the kd-tree. (Recall that each point of the data set is represented as a coordinate vector in $\mathbb{R}^d$.)

$\mathbf{s}(u)$: *weighted centroid*; contains the vector sum of the points associated with this node.

$ss(u)$: *sum of squares*; contains the sum of the dot products $(\mathbf{x} \cdot \mathbf{x})$ for all points $\mathbf{x}$ associated with this node.

$w(u)$: *weight*; contains the number of points associated with this node.

The above quantities can be computed in $O(dn)$ time by a simple postorder traversal of the kd-tree. We omit the straightforward details. The following lemma shows that once the set of nodes associated with a given center are known, the centroid of the set and the distortion of the resulting cluster can be computed.

Lemma 3.1 *Consider a fixed cluster S_j, and let $U = \{u_1, u_2, \ldots, u_m\}$ be a set of nodes that are assigned to this cluster, so that S_j is the disjoint union of $S(u_i)$, for $1 \leq i \leq m$. Consider the following sums of the above quantities associated with the nodes in U:*

$$\mathbf{s}_j = \sum_{i=1}^{m} \mathbf{s}(u_i), \qquad ss_j = \sum_{i=1}^{m} ss(u_i), \qquad w_j = \sum_{i=1}^{m} w(u_i).$$

Then the size of the cluster is $n_j = w_j$, the centroid of the cluster is $(1/n_j)\mathbf{s}_j$, and the average distortion of the cluster is

$$\Delta_j^{(2)} = \frac{1}{w_j} ss_j - \frac{2}{w_j}(\mathbf{z}_j \cdot \mathbf{s}_j) + (\mathbf{z}_j \cdot \mathbf{z}_j).$$

Proof: Because $\bigcup_{i=1}^{m} S(u_i)$ is a disjoint partition of S_j the following identities hold:

$$\mathbf{s}_j = \sum_{\mathbf{x}\in S_j} \mathbf{x}, \qquad ss_j = \sum_{\mathbf{x}\in S_j} (\mathbf{x}\cdot\mathbf{x}), \qquad w_j = \sum_{\mathbf{x}\in S_j} 1 = |S_j| = n_j.$$

The first two claims follow directly from these identities, leaving only the expression of the average distortion to prove. In a slight abuse of notation, for two vectors $\mathbf{x}$ and $\mathbf{z}$, we express their dot products as $\mathbf{x}^2 = (\mathbf{x}\cdot\mathbf{x})$ and $\mathbf{xz} = (\mathbf{x}\cdot\mathbf{z})$. Then we can express the total distortion for the jth cluster as:

$$\begin{aligned} n_j\Delta_j^{(2)} &= \sum_{\mathbf{x}\in S_j} (\mathbf{x}-\mathbf{z}_j)^2 = \sum_{\mathbf{x}\in S_j} (\mathbf{x}^2 - 2\mathbf{x}\mathbf{z}_j + \mathbf{z}_j^2) = \sum_{\mathbf{x}\in S_j} \mathbf{x}^2 - \sum_{\mathbf{x}\in S_j} 2\mathbf{x}\mathbf{z}_j + \sum_{\mathbf{x}\in S_j} \mathbf{z}_j^2 \\ &= \sum_{\mathbf{x}\in S_j} (\mathbf{x}\cdot\mathbf{x}) - 2\left(\mathbf{z}_j \cdot \sum_{\mathbf{x}\in S_j} \mathbf{x}\right) + w_j(\mathbf{z}_j\cdot\mathbf{z}_j) \\ &= ss_j - 2(\mathbf{z}_j\cdot\mathbf{s}_j) + w_j(\mathbf{z}_j\cdot\mathbf{z}_j). \end{aligned}$$

The final result follows by dividing by $n_j = w_j$. □

3.4 Assigning Nodes to Centers

All that remains is to explain how the filtering algorithm assigns nodes to each of the cluster centers. Recall that the input to the algorithm is the set S given in the form of a kd-tree, the statistical quantities $\mathbf{s}(u)$, $ss(u)$, and $w(u)$ for each node u of the kd-tree, and the locations of the cluster centers $\mathbf{z}_j$. As the algorithm assigns a node u to a center $\mathbf{z}_j$, it adds these three quantities to the associated sums $\mathbf{s}_j$, ss_j, and w_j, as defined in the proof of Lemma 3.1. Upon termination of the algorithm, each center $\mathbf{z}_j$ is associated with the sum of these quantities for all the points S_j.

As mentioned above, the filtering algorithm visits the nodes of the tree in a recursive top-down manner. For each node it visits, it maintains the subset of centers, called *candidates*, such that the closest center to any point in the node's cell is one of these candidate centers. Thus, for each node we keep track of a subset of centers that may serve as the nearest center for any point within the cell. Unfortunately, we know of no sufficiently efficient test to determine the set of true candidates (which involves determining the set of Voronoi cells overlapped by an axis-aligned rectangle). Instead, we will describe a simple procedure that associates each node with a superset of its true candidates.

To start the process off, the candidates for the root node of the kd-tree consists of all k centers. The centers are then filtered through the kd-tree as follows. Let C be the cell associated with the current node u, and let Z be the set of the candidate centers associated with C. First, the closest center $\mathbf{z}^* \in Z$ to the midpoint of C is computed. Then, for the rest of the candidates $\mathbf{z} \in Z \backslash \mathbf{z}^*$, if all parts of C are farther from $\mathbf{z}$ than they are to $\mathbf{z}^*$, we may conclude that $\mathbf{z}$ cannot serve as the nearest center for any point in u. So we can eliminate, or *filter*, $\mathbf{z}$ from the set of candidates. If there is only one candidate center (that is, $|Z| = 1$), then the node in question is assigned to this center. In particular, this means that the quantities $\mathbf{s}$, ss, and w for node u are added to the corresponding sums for this center. Otherwise, for an internal node, we pass the surviving set of candidates to its two children, and repeat the process recursively. If the algorithm reaches a leaf node having two or more candidates, the distances from all centers of Z to the node's data point are calculated, and this data point is assigned to the nearest candidate center.

In order to determine whether any part of C is closer to candidate $\mathbf{z}$ than to $\mathbf{z}^*$ we proceed as follows. Let H be the hyperplane bisecting the line segment $\overline{\mathbf{z}\mathbf{z}^*}$ (see Fig. 3). We can filter $\mathbf{z}$ if C is entirely on the same side of H as $\mathbf{z}^*$. This condition is tested through the use of a vector $\mathbf{w} = \mathbf{z} - \mathbf{z}^*$, from $\mathbf{z}^*$ to $\mathbf{z}$. Let $\mathbf{v}$ be the vertex of C that maximizes the dot product $(\mathbf{v} \cdot \mathbf{w})$, that is $\mathbf{v}$ is farthest vertex in C in direction of $\mathbf{w}$. If $\text{dist}(\mathbf{z}, \mathbf{v}) \geq \text{dist}(\mathbf{z}^*, \mathbf{v})$, then $\mathbf{z}$ is pruned. The choice of the vertex $\mathbf{v}$ can be determined simply by the signs of the individual coordinates of $\mathbf{w}$. (See [27] for details.) The process requires $O(d)$ time for each center tested.

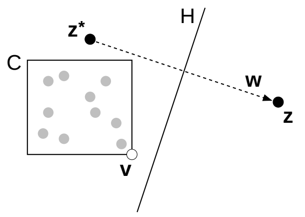


Fig. 3: Filtering process where $\mathbf{z}$ is pruned.

The filtering algorithm achieves its efficiency by assigning many points at once to each center. A straightforward implementation of Lloyd's algorithm requires $O(kn)$ time to compute the distance from each of the n points to each of the k centers. The corresponding measure of complexity for the filtering algorithm is the number of interactions between nodes and candidates. Kanungo *et al.* [27] have shown experimentally that this number is smaller by factors ranging from 10 to 200 for low dimensional clustered data sets. Even with the additional preprocessing time and overhead, the speed-ups in actual CPU time can be quite significant.

3.5 Approximate Filtering

As with many approaches based on spatial subdivision methods, the filtering algorithm suffers from the so-called "curse of dimensionality," which in our context means that as the dimension increases the algorithm's running time increases exponentially as a function of the dimension. This was observed by Kanungo *et al.* in their analysis of the filtering algorithm [27]. The problem with high dimensions stems from the fact that any approach based on kd-trees relies on the hypothesis that the rectangular cell associated with each node is a good approximation to the extent of the subset of points of S that lie within the cell. This is true in when the dimension is low. As the dimension increases, however, the cell progressively becomes a poorer approximation to the set of points lying within it. As a result, the pruning process is less efficient, and more nodes need to be visited by the filtering algorithm before termination.

Our approach for dealing with this problem is to apply filtering in an approximate manner, and so to trade accuracy for speed. In our case, we allow the user to provide a parameter $\epsilon > 0$, and the filtering algorithm is permitted to assign each point of S to any center point that is within a distance of up to $(1 + \epsilon)$ times the distance to the closest center. This makes it easier to prune a cell from further consideration, and thus ameliorates the adverse effects arising in high dimensions.

This can be incorporated into the filtering process as follows. We recall the notation from the previous section, where u is the current node being processed, C and Z denote, respectively, the cell and set of candidate centers associated with u, and $\mathbf{z}^* \in Z$ is the closest center in Z to the midpoint of C. Our goal is to determine those centers $\mathbf{z} \in Z \backslash \{\mathbf{z}^*\}$, such that for every center $\mathbf{x} \in C$ we have $\|\mathbf{x}\mathbf{z}^*\| \leq (1+\epsilon)\|\mathbf{x}\mathbf{z}\|$. All such center points $\mathbf{z}$ can be filtered. In geometric terms, this is equivalent to replacing the bisector test used in the exact algorithm with a test involving an approximate bisector, denoted $H_\epsilon(\mathbf{z}, \mathbf{z}^*)$. The latter is defined to be the set of points $\mathbf{x}$, such that $\|\mathbf{x}\mathbf{z}^*\| = (1+\epsilon)\|\mathbf{x}\mathbf{z}\|$. (See Fig. 4.)

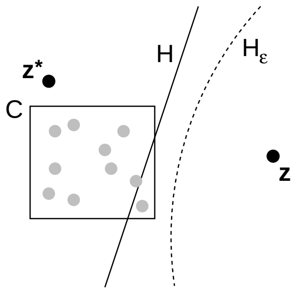


Fig. 4: Approximate filtering, where $\mathbf{z}$ is pruned.

The hyperplane bisector test of the previous section must be adapted to determine whether C is stabbed by $H_\epsilon(\mathbf{z}, \mathbf{z}^*)$. At first, this might seem to be a much harder test to perform. For example, it is no longer sufficient to merely test an appropriate vertex of C, since it is possible that the approximate bisector intersects the interior of a facet of C, while all the vertices lie to one side of the approximate bisector. What saves the day is the fact that the approximate bisector is a hypersphere, and hence the problem reduces to computing the distance between an axis-aligned rectangle and the center of this hypersphere, which can be computed easily. For completeness, we present the following two technical lemmas, which provide the necessary groundwork.

Lemma 3.2 *Given $\epsilon > 0$, and two points $\mathbf{z}$ and $\mathbf{z}^*$ in d-space, $H_\epsilon(\mathbf{z}, \mathbf{z}^*)$ is a $(d-1)$-sphere of radius r_ϵ centered at the point $\mathbf{c}_\epsilon$, where*

$$r_\epsilon = \frac{1+\epsilon}{\gamma-1}\|\mathbf{z}\mathbf{z}^*\| \qquad \textit{and} \qquad \mathbf{c}_\epsilon = \frac{1}{\gamma-1}(\gamma\mathbf{z} - \mathbf{z}^*), \qquad \textit{where} \qquad \gamma = (1+\epsilon)^2.$$

Proof: A point $\mathbf{x}$ lies on H_ϵ if and only if

$$\|\mathbf{x}\mathbf{z}^*\|^2 = (1+\epsilon)^2\|\mathbf{x}\mathbf{z}\|^2.$$

As before, it will be convenient to express dot products using $\mathbf{x}^2 = (\mathbf{x} \cdot \mathbf{x})$ and $\mathbf{x}\mathbf{z} = (\mathbf{x} \cdot \mathbf{z})$. The above is equivalent to

$$\begin{aligned} (\mathbf{x} - \mathbf{z}^*)^2 &= (1+\epsilon)^2(\mathbf{x} - \mathbf{z})^2 \\ \mathbf{x}^2 - 2\mathbf{x}\mathbf{z}^* + \mathbf{z}^{*2} &= \gamma(\mathbf{x}^2 - 2\mathbf{x}\mathbf{z} + \mathbf{z}^2). \end{aligned}$$

Expanding and completing the square yields

$$\begin{aligned} (\gamma-1)\mathbf{x}^2 - 2(\gamma\mathbf{z} - \mathbf{z}^*)\mathbf{x} + (\gamma\mathbf{z}^2 - \mathbf{z}^{*2}) &= 0 \\ \mathbf{x}^2 - \frac{2}{\gamma-1}(\gamma\mathbf{z} - \mathbf{z}^*)\mathbf{x} + \frac{1}{(\gamma-1)^2}(\gamma\mathbf{z} - \mathbf{z}^*)^2 &= \frac{1}{(\gamma-1)^2}(\gamma\mathbf{z} - \mathbf{z}^*)^2 - \frac{1}{\gamma-1}(\gamma\mathbf{z}^2 - \mathbf{z}^{*2}) \\ \left(\mathbf{x} - \frac{1}{\gamma-1}(\gamma\mathbf{z} - \mathbf{z}^*)\right)^2 &= \frac{1}{(\gamma-1)^2}(\gamma\mathbf{z} - \mathbf{z}^*)^2 - \frac{1}{\gamma-1}(\gamma\mathbf{z}^2 - \mathbf{z}^{*2}). \end{aligned}$$

The left-hand side is $(\mathbf{x} - c_\epsilon)^2$. Expanding the right-hand side gives

$$\begin{aligned}
(\mathbf{x} - c_\epsilon)^2 &= \frac{1}{(\gamma-1)^2}((\gamma\mathbf{z} - \mathbf{z}^*)^2 - (\gamma - 1)(\gamma\mathbf{z}^2 - \mathbf{z}^{*2})) \\
&= \frac{1}{(\gamma-1)^2}((\gamma^2\mathbf{z}^2 - 2\gamma\mathbf{z}\mathbf{z}^* + \mathbf{z}^{*2}) - (\gamma^2\mathbf{z}^2 - \gamma\mathbf{z}^{*2} - \gamma\mathbf{z}^2 + \mathbf{z}^{*2})) \\
&= \frac{1}{(\gamma-1)^2}(\gamma\mathbf{z}^2 - 2\gamma\mathbf{z}\mathbf{z}^* + \gamma\mathbf{z}^{*2}) \\
&= \frac{\gamma}{(\gamma-1)^2}(\mathbf{z} - \mathbf{z}^*)^2 \;=\; \left(\frac{1+\epsilon}{\gamma-1}\|\mathbf{z}\mathbf{z}^*\|\right)^2 \;=\; r_\epsilon^2.
\end{aligned}$$

This is the equation of the desired hypersphere. □

Lemma 3.3 *The closest (Euclidean) distance between an axis-aligned hyperrectangle in $\mathbb{R}^d$ and any point $\mathbf{c} \in \mathbb{R}^d$ can be computed in $O(d)$ time.*

Proof: Let $\mathbf{v} = (v_1, \ldots, v_d)$ and $\mathbf{w} = (w_1, \ldots, w_d)$ be the rectangle vertices with the lowest and highest coordinate values, respectively. (For example, these would be the lower left and upper right vertices in the planar case.) The rectangle is just the d-fold intersection of axis-orthogonal strips

$$\{(x_1, \ldots, x_d) \mid v_i \le x_i \le w_i\}.$$

Based on the location of $\mathbf{c}$ relative to each of these strips, we can compute the squared distance from $\mathbf{c} = (c_1, \ldots, c_d)$ to the rectangle as $\sum_{i=1}^d \delta_i^2$, where

$$\delta_i = \begin{cases} v_i - c_i & \text{if } c_i < v_i \\ 0 & \text{if } v_i \le c_i \le w_i \\ c_i - w_i & \text{if } w_i < c_i. \end{cases}$$

The final distance is the square root of this sum. □

Using these two lemmas, it is now easy to see how to replace the exact filtering step described in the previous section with an approximate filtering test, which also runs in $O(d)$ time. Given candidate centers $\mathbf{z}$ and $\mathbf{z}^*$, we apply Lemma 3.2 to compute r_ϵ and $\mathbf{c}_\epsilon$. We then apply Lemma 3.3 to compute the closest distance between the cell C and $\mathbf{c}_\epsilon$. If this distance is greater than r_ϵ, then $\mathbf{z}$ is pruned. The remainder of the algorithm is the same. In Section 5.3 below, we present experimental evidence for the benefits of using approximate filtering.

Although points are assigned to cluster centers that are ϵ-nearest neighbors, it does not follow that the result produced by the approximate version of the ISOCLUS algorithm results in an ϵ-approximation in the sense of distortion. The reason is that ISOCLUS is a heuristic and does not provide any guarantees on the resulting distortion. It follows some path in the space of possible solutions to some local minimum. Even a minor change to the algorithm's definition can alter this path, and may lead to a local minimum of a significantly different value, either larger or smaller.

4 Our Modifications and Improvements

As mentioned earlier, most of the computational effort in the ISOCLUS algorithm is spent calculating and updating distances and distortions in Steps 2–5. These steps take $O(kn)$ time, whereas all the other steps can be performed in $O(k)$ time, where k is the current number of centers. Our improvement is achieved by adapting the filtering algorithm to compute the desired information. This is the reason for computing the additional statistical information, which was described in the previous section.

There is one wrinkle, however. The filtering algorithm achieves its efficiency by processing points in groups, rather than individually. This works fine as long as the statistical quantities being used by the algorithm can be computed in an aggregated manner. This is true for the centroid, as shown in Lemma 3.1, since it involves the sum of coordinates. Generally, the filtering method can be applied to any polynomial function of the point and center coordinates. However, there is one statistical quantity computed by the ISOCLUS algorithm that does not satisfy this property. In particular, Step 5 of the ISOCLUS algorithm involves computing the sum of Euclidean distances from each point to its closest center as a measure of the dispersion of the cluster. This information is used later in Step 8 to determine whether to split the cluster. This involves computing the sum of square roots, and we know of no way to aggregate this processing.

Rather than implementing ISOCLUS exactly as described in [37], we modified Step 5 as follows. For each cluster j, instead of computing the average Euclidean distance of each point to its center, Δ_j, we compute the average *squared Euclidean distance*, denoted $\Delta_j^{(2)}$. In order to preserve the metric units, we use the square root of this quantity, denoted Δ'_j. In short, we modified the definitions of Step 5 by computing the following quantities:

$$\begin{aligned}
\Delta_j^{(2)} &= \frac{1}{n_j}\sum_{\mathbf{x}\in S_j}\|\mathbf{x}-\mathbf{z}_j\|^2, \qquad \text{for } 1\le j\le k \\
\Delta'_j &= \sqrt{\Delta_j^{(2)}}, \qquad \text{for } 1\le j\le k \\
\Delta' &= \frac{1}{n}\sum_{j=1}^{k} n_j\Delta'_j.
\end{aligned}$$

The decision as to whether to split a cluster in Step 8 depends on the relative sizes of Δ'_j and Δ', rather than Δ_j and Δ. Note that this can produce different results. Nonetheless, having experimented with both synthetically generated data and real images, we observed that the actual performance of our algorithm was quite similar to that of ISOCLUS, in terms of the number of clusters obtained and the positions of their centers. This will be demonstrated in the next section. Thus, we believe that this modification does not significantly alter the nature of the algorithm, and has the benefit of running significantly faster. The value Δ'_j can be computed as outlined in Lemma 3.1. In the next section we present the experimental results obtained using our convention, and in Section 6 we provide theoretical justification for the modifications made.

5 Experiments

In order to establish the efficiency of both our new exact and approximate algorithmic versions, and to determine the degree of similarity in clustering performance with the existing ISOCLUS algorithm, we ran a number of experiments on synthetic data, as well as remotely sensed images. Our modified algorithm involves changing both the functionality and computational approaches. To make the comparisons clearer, we implemented an intermediate, or *hybrid*, algorithm, which is functionally equivalent to one variant but uses the same computational approach as the other.

Standard version (Std): The straightforward implementation of ISOCLUS as described in [37], which uses average Euclidean distances in Step 5 and Step 8.

Hybrid version (Hyb): A modification of the standard version using Δ'_j and Δ' rather than Δ_j and Δ in Step 5 and Step 8, but without using the filtering algorithm.

Filtering version (Fil): The same modification, but using the filtering algorithm for greater efficiency.

The Hybrid and Filtering versions are functionally equivalent, but use different computational approaches. The Standard and Hybrid versions are roughly equivalent in terms of the computational methods, but are functionally distinct. Our goal is to show that the Standard and Hybrid versions are nearly functionally equivalent, and that in many instances the Filtering version is significantly more efficient. All experiments were run on a SUN Blade 100 running Solaris 2.8, using the g++ compiler (version 2.95.3).

We mention for completeness that we also implemented and tested a fourth version, the results of which are not reported, as they were not competitive with the filtering algorithm. The latter variant stores the k center points in a kd-tree, as implemented in the ANN library [3]. The nearest center to each data point is then computed by a search of this tree. This approach proved to be consistently slower than the filtering algorithm for two reasons. First, there are significantly fewer center points than query points ($k \ll n$). Thus, there are lower savings in running time that would result by storing the k center points in a tree as compared to the savings that result by storing the n data points in a tree. Second, the center points change with each iteration, and so the tree would need to be rebuilt constantly.

The remainder of this section is devoted to presenting the results of the various experiments we ran. Section 5.1 presents the performance of these algorithms on synthetically generated clustered data sets of various sizes and in various dimensions. In Section 5.2 we present experiments on data sets generated from an application in remote sensing, in which ISOCLUS is regularly used. Next, in Section 5.3 we investigate the performance of the approximate version of the filtering algorithm. Finally, in Section 5.4 we consider the effect of increasing the dimension of the data set on the running time and speed-up, for both the exact and approximate versions.

5.1 Synthetic Data

We ran the following three sets of experiments on synthetically generated data sets to analyze the performance of our algorithm. All experiments were run in dimensions 3, 5, and 7. (This choice of dimensions was guided by the fact that many applications of ISOCLUS in remote sensing involve Landsat satellite image data. Raw Landsat data contains 7 spectral bands, and reductions to dimensions 3 and 5 are quite common.)

(1) For the first set of experiments we generated $n = 10,000$ data points. In each case the points were sampled with equal probability from a variable number of Gaussian clusters ranging from 10 to 100, by a method described below.

(2) In the second set of experiments five data sets were considered, containing 100, 500, 1000, 5000, and $10,000$ points, respectively. In each case the points were distributed evenly among 20 Gaussian clusters.

(3) In the third set of experiments, we varied both the number of randomly generated points and the number of clusters. Specifically, we considered data sets containing 100, 500, 1000, 5000, and $10,000$ points. For each data set, the points were distributed evenly among 5, 10, 20, 40, and 80 Gaussian clusters.

All of the above experiments involved points drawn from a collection of some number k of Gaussian clusters. This was done as follows. Cluster centers were sampled uniformly at random from the hypercube $[-1,1]^d$ of side length 2. In order to generate a point for each cluster, a vector was generated, such that each of its coordinates was drawn from a Gaussian distribution with a given standard deviation $\sigma = (1/k)^{1/d}$.

The value of σ was derived by the following reasoning. In order for the results to be comparable across different dimensions and with different numbers of clusters, it is desirable that clusters have comparable degrees of overlap. In low dimensions, a significant amount of the probability mass of a Gaussian cluster lies within a region whose volume is proportional to $(2\sigma)^d$. We wish to subdivide a cube of unit volume uniformly into k clusters, which suggests that each cluster should cover $1/k$-th of the total volume, and hence σ should be chosen such that $(2\sigma)^d = 2^d/k$, from which the above value of σ was obtained.

We ran the ISOCLUS algorithms for a maximum of 20 iterations ($I_{\max} = 20$). In each case the initial number of clusters was set to the actual number of clusters generated ($k_{\text{init}} = k$), the maximum cluster standard deviation was set to twice the standard deviation of the distribution ($\sigma_{\max} = 2\sigma$), and the minimum cluster separation was set to 0.001 ($L_{\min} = 0.001$). We decided to remove a cluster if it contained fewer than 1/5 of the average cluster size, and so set $n_{\min} = n/(5k_{\text{init}})$. For the first set of experiments where $n = 10,000$ was fixed, we set the initial number of clusters to 10, 20, 40, 80, and 100, in accordance with the respective number of actual clusters generated. In each case, the results were averaged over five runs. The results of the above 3 sets of experiments are shown in Table 1, Table 2, and Table 3.

For each run, we computed the running time in CPU seconds, the final number of centers, and the final average distortion. Not surprisingly, since the hybrid and filtering versions implement the same functional specifications, the final numbers of centers and final distortions obtained were almost identical. (Small differences were observed due to floating point round-off errors.) Thus, we listed together the corresponding results in the tables (under "Hyb/Fil"). We also computed the *speed-up*, which is defined as the ratio between the CPU time of the hybrid version and that of the filtering version.

In support of our claim that using squared distances does not significantly change the algorithm's clustering performance, observe that both algorithms performed virtually identically with respect to average distortions and the final number of centers. Also observe that the standard and hybrid versions ran in roughly the same time, whereas the filtering version ran around 1.3 to 15.2 times faster than the other two. Fig. 5 shows our experimental results on the synthetic data sets. We can see that for a fixed number of points, increasing the number of clusters increases both the CPU time and speed-up. The same result holds when we increase the number of points and fix the other parameters.

Table 1: Results for Synthetic Data with n = 10,000

Dim	k_{init}	Final Centers		Avg. Distortion		CPU Seconds			Speed-up
		Std	Hyb/Fil	Std	Hyb/Fil	Std	Hyb	Fil	
3	10	10	10	0.385	0.385	5.00	5.14	1.420	3.62
	20	20	20	0.286	0.286	8.74	9.07	1.788	5.07
	40	40	40	0.120	0.120	16.39	17.20	2.022	8.50
	80	78	78	0.077	0.077	33.87	34.99	2.626	13.32
	100	100	100	0.061	0.061	43.34	44.95	2.956	15.21
5	10	9	9	2.108	2.108	6.86	6.77	3.092	2.189
	20	19	19	1.184	1.184	12.58	13.20	3.880	3.403
	40	36	36	0.819	0.819	21.79	22.83	5.372	4.249
	80	79	79	0.490	0.490	48.69	50.01	7.998	6.253
	100	93	93	0.478	0.478	57.67	59.30	9.172	6.465
7	10	10	10	4.062	4.062	9.18	9.38	5.29	1.771
	20	20	20	1.971	1.971	16.31	16.82	7.40	2.274
	40	32	32	2.303	2.303	26.11	26.59	11.50	2.312
	80	74	74	1.447	1.447	59.27	60.29	22.07	2.732
	100	93	93	1.242	1.242	75.27	76.55	26.02	2.942

Table 2: Results for Synthetic Data with $k_{\text{init}} = 20$

Dim	n	Final Centers		Avg. Distortion		CPU Seconds			Speed-up
		Std	Hyb/Fil	Std	Hyb/Fil	Std	Hyb	Fil	
3	100	20	20	0.164	0.164	0.100	0.088	0.048	1.833
	500	20	20	0.278	0.278	0.446	0.428	0.148	2.892
	1000	20	20	0.265	0.265	0.902	0.876	0.256	3.422
	5000	20	20	0.288	0.288	4.512	4.232	0.960	4.408
	10000	20	20	0.286	0.286	9.290	8.804	1.770	4.974
5	100	20	20	0.828	0.828	0.138	0.116	0.092	1.261
	500	17	17	1.095	1.095	0.560	0.556	0.286	1.944
	1000	20	20	1.074	1.074	1.368	1.300	0.600	2.167
	5000	19	19	1.188	1.188	6.304	6.130	2.234	2.744
	10000	19	19	1.184	1.184	12.958	12.812	3.868	3.312
7	100	20	20	1.349	1.349	0.168	0.158	0.112	1.411
	500	17	17	1.957	1.957	0.690	0.692	0.478	1.450
	1000	18	18	1.990	1.990	1.546	1.526	0.924	1.652
	5000	19	19	1.990	1.990	8.078	7.994	4.146	1.928
	10000	20	20	1.971	1.971	16.740	16.604	7.472	2.222

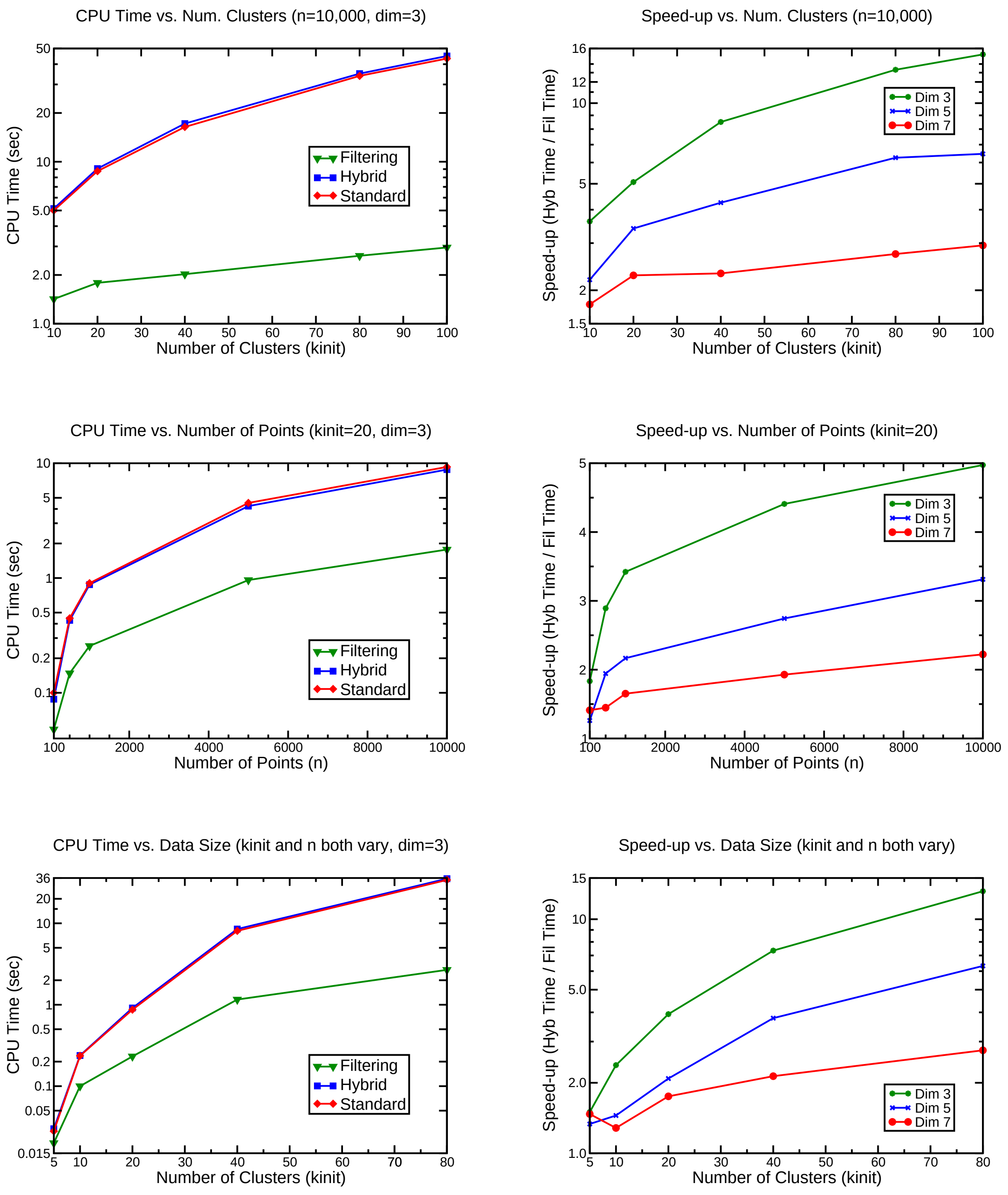


Fig. 5: CPU times and speed-ups for the various algorithms run on synthetic data. (Note that the x and y axes do not intersect at the origin.) For the bottom pair of plots, note that n also varies with k_{init} as indicated in Table 3.

Table 3: Results for Synthetic Data where Both n and k_{init} Vary

Dim	n	k_{init}	Final Centers		Avg. Distortion		CPU Seconds			Speed-up
			Std	Hyb/Fil	Std	Hyb/Fil	Std	Hyb	Fil	
3	100	5	5	5	0.480	0.480	0.028	0.03	0.020	1.500
	500	10	10	10	0.357	0.357	0.236	0.24	0.100	2.380
	1000	20	20	20	0.265	0.265	0.870	0.91	0.232	3.931
	5000	40	40	40	0.120	0.120	8.082	8.50	1.158	7.344
	10000	80	78	78	0.077	0.077	34.074	35.33	2.682	13.174
5	100	5	5	5	2.350	2.350	0.036	0.04	0.030	1.334
	500	10	8	8	2.095	2.095	0.280	0.29	0.200	1.450
	1000	20	20	20	1.074	1.074	1.280	1.27	0.608	2.086
	5000	40	39	39	0.797	0.797	11.904	12.12	3.208	3.778
	10000	80	79	79	0.490	0.490	48.470	50.49	7.994	6.316
7	100	5	4	4	4.417	4.417	0.042	0.05	0.03	1.471
	500	10	9	9	4.321	4.321	0.398	0.42	0.33	1.282
	1000	20	18	18	1.990	1.990	1.550	1.58	0.90	1.750
	5000	40	36	36	2.201	2.201	14.782	15.07	7.06	2.135
	10000	80	74	74	1.447	1.447	46.966	60.69	22.04	2.753

5.2 Image Data

For image data we used two different data sets from remotely sensed imagery: A Landsat data set and a MODIS scene. For the Landsat data we ran nine tests on a 256×256 image of Ridgely, Maryland ($n = 65,536$). The first set of experiments involved three tests on 3-dimensional data using spectral bands 3, 4, and 5. The initial number of clusters was set to 10, 50, and 100. This choice covers the range of values used in typical remote sensing applications. The second set of experiments was performed in 5-dimensional space using spectral bands 3 through 7, and the third set was carried out in 7-dimensional space using all seven bands. The tests in dimensions 5 and 7 were performed with 10, 50, and 100 initial centers (k_{init}), as well. We ran all nine tests with the three versions of ISOCLUS, each for 20 iterations, $\sigma_{\max} = 15$, $L_{\min} = 10$, and $n_{\min} = n/(5k_{\text{init}})$ (approximately), and k_{init} of 10, 50, and 100. Each experiment was run 10 times, invoking every time the algorithmic version in question with a different set of initial random centers. The results obtained were averaged over these 10 runs. (This accounts for the noninteger number of "Final Centers" reported in the tables.)

The results are summarized in Table 4. As with the tests on synthetic data, all versions performed essentially equivalently with respect to the number of centers and final distortions. The filtering version was faster by a factor of roughly 4 to 30. Fig. 6 shows the original data and the clustered images obtained due to the standard and filtering ISOCLUS in 3-dimensional space. (As indicated, the clusters for the two versions were essentially identical.)

For the MODIS data set we repeated the above three sets of experiments on a 128×128 ($n = 16,384$) subimage acquired over an agricultural area from the Konza Prairie in Kansas. The results are summarized in Table 5. As with the Landsat data set, we experimented in dimensions 3, 5, and 7, only that here the spectral bands were selected through principal component analysis (PCA) by the standard approach based on the Karhunen-Loéve transformation [13].

Table 4: Results for Landsat Data Set

Dim	k_{init}	Final Centers		Avg. Distortion		CPU Seconds			Speed-up
		Std	Hyb/Fil	Std	Hyb/Fil	Std	Hyb	Fil	
	10	6.3	6.3	67.92	67.86	28.109	27.370	5.838	4.688
3	50	10.1	9.9	43.49	44.11	84.729	82.213	7.182	11.447
	100	22.1	22.9	25.31	24.55	290.110	280.470	9.117	30.763
	10	5.9	5.9	144.04	144.04	43.989	43.169	10.352	4.170
5	50	15.6	15.7	85.50	91.02	174.590	171.160	17.778	9.628
	100	23.9	22.7	33.93	35.12	367.130	359.200	18.748	19.159
	10	7.3	7.3	169.17	169.17	62.214	61.277	15.788	3.881
7	50	15.8	15.8	107.68	107.68	206.720	203.610	26.659	7.638
	100	22.1	22.1	46.21	46.21	442.650	430.860	31.655	13.611

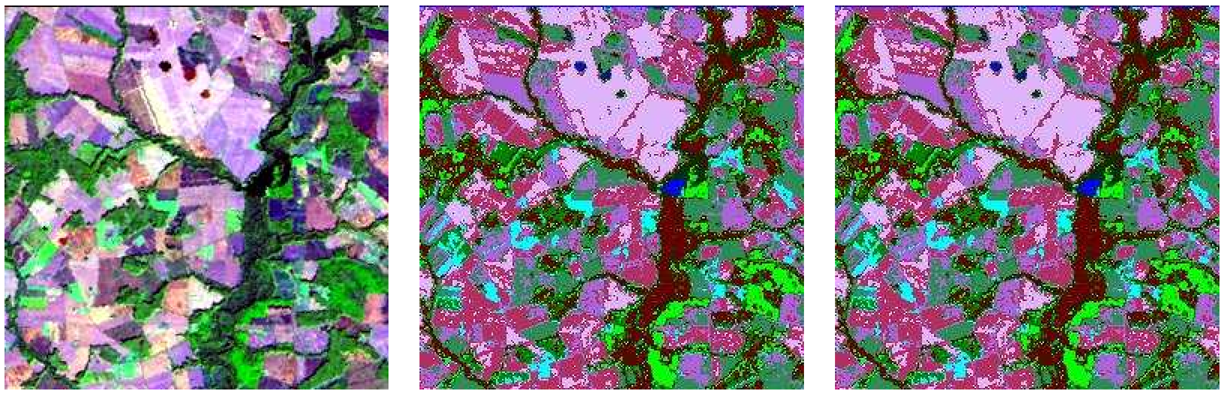


Fig. 6: A Landsat scene and its clustered images: (a) 256×256 Landsat image of Ridgely, Maryland (bands 3, 4, and 5), (b) clustered image due to standard ISOCLUS, and (c) clustered image due to the Filtering variant.

The initial number of clusters experimented with in each case was 10, 50, and 100. The remaining parameters used were essentially the same as those for the Landsat data set, except for $n_{\min} = 45$. As before, each experiment was repeated 10 times, invoking the algorithm in question every time with a different set of initial random centers. The results reported were averaged over these 10 runs.

Table 5: Results for the MODIS Data Set

Dim	k_{init}	Final Centers		Avg. Distortion		CPU Seconds			Speed-up
		Std	Hyb/Fil	Std	Hyb/Fil	Std	Hyb	Fil	
3	10	17.2	17.8	389.69	383.46	14.515	14.577	2.231	6.534
	50	51.7	51.7	177.95	177.94	38.562	37.209	2.784	13.365
	100	98.4	98.4	114.86	114.87	83.172	80.444	3.600	22.345
5	10	20.3	21.0	970.00	946.91	22.761	21.901	4.345	5.041
	50	69.5	69.5	478.09	478.09	82.020	79.916	7.450	10.727
	100	116	116.0	372.55	372.56	143.360	139.910	9.805	14.269
7	10	20.8	20.8	1437.30	1443.00	28.506	28.360	6.454	4.394
	50	79.5	81.2	728.53	722.13	143.950	144.740	16.950	8.539
	100	134.9	134.5	564.94	565.80	255.950	252.320	24.393	10.344

The final results in dimensions 3 and 5 were identical with respect to both the final number of clusters and the final distortions. In dimension 7, while all versions of the algorithm resulted in an (almost) identical final number of clusters, their distortions were slightly different. The filtering version was faster by factors ranging from roughly 4 to 22. The speed-ups were most dramatic for the cases involving a large numbers of clusters. This is to be expected because the filtering algorithm achieves its improvement by eliminating unpromising candidate centers from consideration.

5.3 Experiments with Approximate Filtering

In order to better understand the effect of approximation, we experimented with the approximate version of the filter-based algorithm. Recall that the algorithm differs from the exact algorithm in how candidate centers are pruned from each node in the process of determining which center is closest to the points of a node. The user supplies a value $\epsilon > 0$, and the algorithm may assign a point to a center whose distance (from the point) is (up to) $(1 + \epsilon)$ times the point's distance to its true nearest center. We performed experiments on both synthetic and satellite image data. In all cases, we ran experiments with approximation parameter $\epsilon \in \{0.1, 0.2, 0.5, 1.0, 1.5\}$, and compared the results against the exact $(\epsilon = 0)$ case. Note that approximation was used in all but the last iteration of the algorithm, in which case exact pruning was performed. The reason is that when the algorithm terminates, we want all the points to be assigned to their true closest center.

The use of ϵ values greater than 1 may seem to be unreasonably large for practical purposes, since this allows for more than 100% relative error. But note that the ϵ value is merely an upper bound on the error committed for each individual point-to-center assignment, and the aggregated effect of these errors is subject to cancelation and may be much smaller. As we shall see below, even for fairly large values of ϵ, the observed distortions relative to the exact version of ISOCLUS were almost always less than 5%.

As mentioned at the end of Section 3.5, ISOCLUS is a heuristic and not an optimization algorithm. Thus, minor changes to the algorithm can result in convergence to local minima with significantly different average distortions. This can happen even when $\epsilon = 0$, because the algorithm is invoked with random initial center points. For this reason, all of the results were averaged over the number of invocations of the algorithm.

For synthetic data, we generated five random sets of $n = 10,000$ points in dimensions 3, 5, and 7. Points were sampled with equal probability from 100 Gaussian clusters with uniformly distributed centers. The distributions and program parameter settings were the same as for the experiments on synthetic data of Section 5.1. We measured the CPU time, the final distortion, and the final number of clusters in each experiment. Finally, we evaluated the algorithm's relative performance with respect to the standard version of ISOCLUS (by invoking the latter on the same data sets). We computed (average) speed-ups, as well as relative distortion errors with respect to the standard version. These results are summarized in Table 6.

Table 6: Results on Synthetic Data with Approx. Filtering, $n = 10,000$, and $k_{\text{init}} = 100$

Dim	ϵ	Final Centers		Avg. Dist.×100		CPU Seconds		Speed-up	Rel Dist
		Std	Fil	Std	Fil	Std	Fil		Err %
3	0.0	98.2	98.2	6.09	6.09	43.12	2.72	15.85	0.00
	0.1		98.2		6.09		4.36	9.89	0.00
	0.2		98.4		6.09		3.90	11.06	0.00
	0.5		98.6		6.11		2.88	14.97	0.33
	1.0		98.0		6.35		1.99	21.67	4.27
	1.5		97.4		6.57		1.61	26.78	7.88
5	0.0	94.6	94.6	47.20	47.20	58.44	8.84	6.61	0.00
	0.1		94.6		47.20		17.40	3.36	0.00
	0.2		94.8		47.11		14.39	4.06	-0.19
	0.5		95.6		47.10		9.40	6.22	-0.21
	1.0		96.8		47.58		5.89	9.92	0.81
	1.5		96.6		48.59		4.20	13.91	2.94
7	0.0	93.2	93.2	124.76	124.76	73.70	24.64	2.99	0.00
	0.1		93.2		124.76		48.63	1.52	0.00
	0.2		93.2		124.76		39.79	1.85	0.00
	0.5		93.6		124.62		23.45	3.14	-0.11
	1.0		93.0		126.36		12.19	6.05	1.28
	1.5		93.4		129.04		7.48	9.85	3.43

For the satellite image data, we used the same Landsat and MODIS data sets and the same parameter settings described in Section 5.2. Also, we used the same experimental setup described above (for the approximate version). The results are shown in Table 7 and Table 8 for the Landsat and MODIS data sets, respectively.

The results demonstrate that approximation can result in significant speed-ups. In spite of the relatively large values of ϵ supplied, it is noteworthy that the average error in the final distortion relative to the exact case ("Rel Dist Err %") was dramatically smaller. It never exceeded 8% and was usually less than 3%. The phenomenon of a geometric approximation algorithm performing significantly better on average than the allowable error bound has been observed elsewhere [3]. Since ISOCLUS is a heuristic, it is possible for the approximate version to converge on a better local minimum, and so in some cases the distortion error is actually negative.

Table 7: Results for Landsat Data Set with Approx. Filtering, $k_{\text{init}} = 25$

Dim	ϵ	Final Centers		Avg. Distortion		CPU Seconds		Speed-up	Rel Dist
		Std	Fil	Std	Fil	Std	Fil		Err %
3	0.0	7.9	7.9	56.58	56.85	47.31	6.61	7.16	0.47
	0.1		7.9		58.03		6.54	7.23	2.56
	0.2		7.8		58.05		5.93	7.98	2.60
	0.5		7.9		57.28		5.35	8.84	1.24
	1.0		8.3		54.87		5.12	9.24	-3.02
	1.5		8.5		55.14		5.06	9.35	-2.55
5	0.0	9.7	9.7	114.32	114.43	88.82	14.17	6.27	0.10
	0.1		9.4		115.26		16.45	5.40	0.82
	0.2		9.3		116.82		13.96	6.36	2.19
	0.5		9.5		109.81		9.45	9.40	-3.95
	1.0		9.2		114.52		7.91	11.23	0.17
	1.5		8.5		116.87		7.79	11.40	2.23
7	0.0	10.8	10.9	139.85	137.75	115.51	20.06	5.76	-1.50
	0.1		11.0		135.88		28.36	4.07	-2.83
	0.2		10.8		137.38		24.36	4.74	-1.77
	0.5		11.2		135.47		17.00	6.79	-3.13
	1.0		10.3		137.34		10.86	10.64	-1.79
	1.5		9.0		145.46		9.64	11.98	4.01

Table 8: Results for MODIS Data Set with Approx. Filtering, $k_{\text{init}} = 75$

Dim	ϵ	Final Centers		Avg. Distortion		CPU Seconds		Speed-up	Rel Dist
		Std	Fil	Std	Fil	Std	Fil		Err %
3	0.0	74.9	74.9	137.93	138.04	58.83	3.18	18.50	0.08
	0.1		74.6		138.59		4.44	13.25	0.48
	0.2		74.5		138.74		3.86	15.24	0.59
	0.5		74.3		141.23		2.74	21.47	2.39
	1.0		75.6		143.55		1.90	30.96	4.07
	1.5		76.8		145.28		1.71	34.40	5.33
5	0.0	93.5	93.5	412.43	412.80	112.62	8.76	12.86	0.09
	0.1		93.3		413.88		15.39	7.32	0.35
	0.2		93.2		414.00		12.69	8.87	0.38
	0.5		94.1		412.80		8.39	13.42	0.09
	1.0		98.7		410.71		5.76	19.55	-0.42
	1.5		104.9		401.72		4.59	24.54	-2.60
7	0.0	106.3	106.1	633.54	635.31	180.77	18.63	9.70	0.28
	0.1		105.8		635.66		32.81	5.51	0.33
	0.2		106.3		634.93		27.71	6.52	0.22
	0.5		105.5		636.08		17.52	10.32	0.40
	1.0		110.0		631.15		10.73	16.85	-0.38
	1.5		118.4		618.51		8.60	21.02	-2.37

It is also noteworthy that the approximate algorithm achieved speed-ups of up to one order of magnitude with low average distortion errors throughout the range of parameter values. Note that increasing ϵ did not always lead to a decrease in execution time. This is because of the sensitivity of ISOCLUS to its starting configuration, which further affects the number of iterations and the number of clusters and their structure.

5.4 Dependence on the Dimension

In this section we study the effect of the dimension of the data set on the running times for various versions of our algorithm. Because of their greater sensitivity to the dimension, the filtering and approximate filtering algorithms exhibit poorer speed-ups as the dimension of the data set increases. To investigate this phenomenon more thoroughly, we generated a synthetic data set of 50,000 points (as described in Section 5.1). We ran experiments in various dimensions for the standard, hybrid, filtering, and approximate filtering algorithms. For the approximate version we considered ϵ values ranging from 0 (equivalent to pure filtering) to 2. The dimensions considered range from 2 to 35. Each experiment was run 5 times, invoking each run with a different set of 100 randomly selected centers ($k_{\text{init}} = 100$). The final number of clusters, distortions, and running times were measured and averaged over these 5 runs.

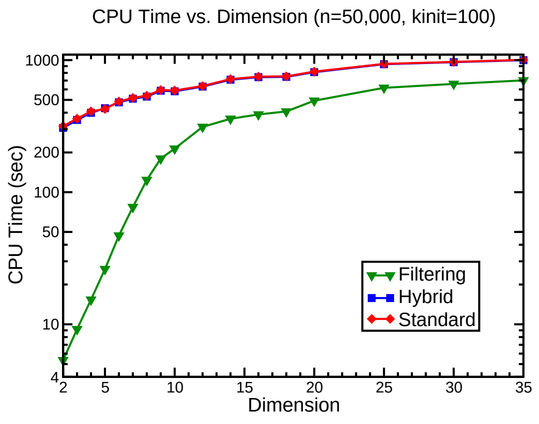
The results for the standard, hybrid, and (exact) filtering algorithms are presented in Table 9 and Fig. 7(a). We see that while filtering yields identical performance to the standard and hybrid versions, in terms of the final number of clusters and distortions, the speed-ups diminish rapidly with the dimension. Nonetheless, it is interesting to note that speed-ups greater than 1 are obtained even for dimensions as high as 35.

Although the exact version of the algorithm exhibited modest speed-ups in higher dimensions, we wanted to find out whether the approximate version could produce still better speed-ups. We repeated the same experiments for the approximate version of the filtering algorithm with ϵ values of 0.5, 1.0, and 2.0. The results show the expected tradeoff, that is, as ϵ increases, the running time tends to decrease while the distortion errors tend to increase. As the dimension increases, nodes are pruned with lower efficiency, and so the algorithm's running time tends to approach that of the exact algorithm. In some cases the running time of the approximate version is even higher than the exact filtering algorithm. This is because the pruning test for the approximate version is computationally more complicated than the pruning test for the exact version. As shown in Fig. 7(b), the approximate filtering algorithm with $\epsilon = 0.5$ is slightly faster than the exact filtering algorithm up to dimension 12. As ϵ increases, the running times improve. For $\epsilon = 1$, the approximate filtering is faster than the exact filtering algorithm up to dimension 20, and for $\epsilon = 2$, the approximate filtering is faster in all of the dimensions tested.

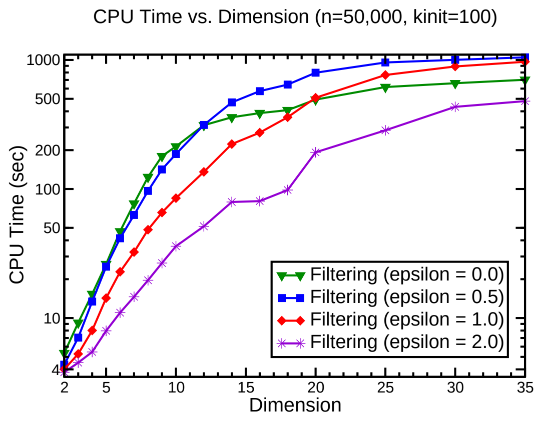
Of course, $\epsilon = 1$ and $\epsilon = 2$ are quite large approximation bounds (allowing for 100% and 200% errors, respectively). For this reason we computed the actual error committed by the algorithm by comparing it with the exact version. Fig. 7(b) and Fig. 8 show that for higher values of ϵ (e.g., $\epsilon = 2$) the average distortion errors are very small, while the speed-ups are quite significant. Remarkably, as the dimension increases, the distortion error becomes successively smaller. Thus, the algorithm obtains significant speed-ups in almost all the dimensions tested with very small actual distortion errors. Unfortunately, the algorithm cannot guarantee small distortion errors for all inputs.

Table 9: Dependence on the Dimension for Synthetic Data, $n = 50,000$, $k_{\text{init}} = 100$

Dim	Final Centers		Avg. Distortion		CPU Seconds			Speed-up
	Std	Hyb/Fil	Std	Hyb/Fil	Std	Hyb	Fil	
2	97.0	97.0	0.007	0.007	316.38	306.74	5.35	57.31
3	98.0	98.0	0.064	0.064	360.30	352.26	9.19	38.33
4	97.8	97.8	0.210	0.210	409.12	398.98	15.36	25.97
5	95.6	95.6	0.483	0.483	426.48	432.40	26.18	16.52
6	96.0	96.0	0.887	0.887	485.88	479.02	46.97	10.20
7	93.8	93.8	1.275	1.275	517.72	510.22	77.18	6.61
8	91.4	91.4	1.985	1.985	537.64	528.96	123.80	4.27
9	92.2	92.2	2.531	2.531	593.90	586.20	179.70	3.26
10	87.2	87.2	2.909	2.909	588.84	580.44	213.80	2.71
12	83.8	83.8	5.208	5.208	636.58	630.06	312.14	2.02
14	84.2	84.2	6.590	6.590	717.56	710.48	359.72	1.98
16	79.4	79.4	8.660	8.660	749.32	742.48	387.52	1.92
18	73.2	73.2	10.892	10.892	753.06	746.50	408.54	1.83
20	73.0	73.0	13.278	13.278	817.90	810.50	492.86	1.64
25	68.8	68.8	19.278	19.278	936.04	931.08	616.90	1.51
30	60.6	60.6	25.650	25.650	968.98	964.26	660.56	1.46
35	54.4	54.4	30.998	30.998	1004.26	998.70	702.30	1.42



(a)



(b)

Fig. 7: CPU times for the various algorithmic versions as a function of the dimension: (a) Standard, hybrid, and exact filtering, and (b) approximate filtering for various ϵ's.

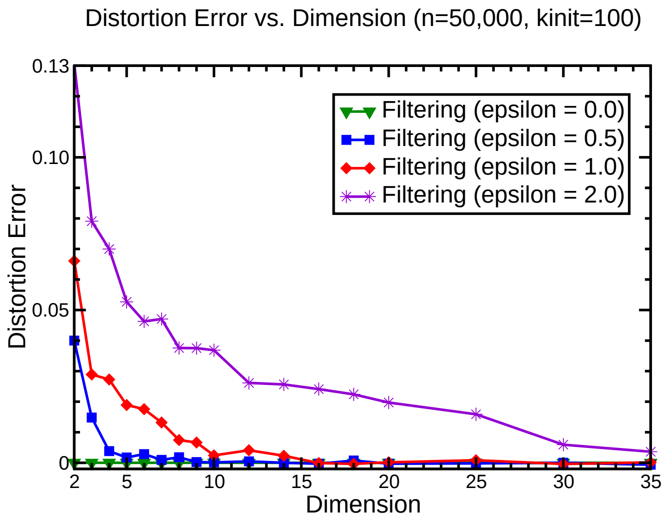


Fig. 8: Average distortion error (relative to the standard version) for the various filtering algorithms as a function of the dimension.

6 Average Distance and Average Distortion

As mentioned, our use of the square root of the average distortion as a measure of cluster dispersion is different from the average distance used in the standard ISOCLUS algorithm. Our experiments suggest that this modification does not make a significant difference in the quality of the resulting clustering. ISOCLUS uses the value of Δ_j in determining whether or not to split a cluster in Step 8. In particular, the jth cluster is split if $\Delta_j > \Delta$. Thus, it would be of interest to establish the conditions for which the following equivalence holds:

$$\Delta_j > \Delta \text{ (in standard ISOCLUS)} \quad \Longleftrightarrow \quad \Delta'_j > \Delta' \text{ (in filtering ISOCLUS)}.$$

This raises an important question as to whether our modification is justifiable, in some sense. To further motivate this question, note that there are other reasonable generalizations of the dispersion that could produce substantially different results.

Had we not considered the square root of the distortion, large distortions would have had a disproportionately greater influence on the average dispersion, which would have resulted in different clusters being split in Step 8 of the algorithm. To see this, consider the following simple 1-dimensional example. We are given three well-separated clusters, each consisting of an equal number of points. The points are drawn from three normal distributions of standard deviations 1, 6, and 9, respectively. Suppose further that the algorithm places three centers, one at the mean of each cluster. If the number of points is large, then the three average Euclidean distances, as computed by the standard version of ISOCLUS, would be close to 1, 6, and 9, respectively. Thus, the overall average would be roughly $\Delta = (1 + 6 + 9)/3 \approx 5.333$, implying that the two clusters with standard deviations of 6 and 9 would be eligible for a split in Step 8 of the algorithm. If squared distances were used instead, however, then the average of the squared distances for each cluster would be very close to 1, 36, and 81, respectively. The overall average would then be $(1 + 36 + 81)/3 \approx 39.333$, implying that only the cluster of standard deviation 9 would be eligible for a split.

An alternative approach involves taking the square root of the average distortion for each cluster (as we do in the filtering algorithm), and then taking the overall average dispersion as the square root of the weighted average of the squared distortions over all the clusters. (This is in contrast to the filtering algorithm, which takes square roots before averaging.) However, this alternative suffers from the same problem as the above approach.

Although it does not seem to be possible to make any worst-case theoretical assertions about the similarity between the results of the standard ISOCLUS algorithm and our modified version, we will endeavor to show that, in the limit, the approach taken in the filtering algorithm does not suffer from the biases of the above alternatives. Our analysis is based on the statistical assumption that points are drawn independently from a number of well separated cluster distributions that are identical up to translation and uniform scaling. This assumption is satisfied in the above examples, where the alternative definitions are shown to fail.

More specifically, we assume that the point set S is drawn from k distinct cluster distributions in $\mathbb{R}^d$. We assume that all the cluster distributions are statistically identical up to a translation and uniform scaling. In particular, let $f(x_1, \ldots, x_d)$ be a d-variate probability density function [10] of the *base cluster distribution*, and let $\mathbf{X}$ denote a random vector sampled from this distribution. Without loss of generality, we may assume that its expected value, $E[\mathbf{X}]$, is the origin. Let $Y = \|\mathbf{X}\|$ be a random variable whose value is the Euclidean length of a vector drawn from this distribution. For the purposes of our analysis, we do not need to make any more specific assumptions about the base distribution. For example, the distribution could be a Gaussian distribution centered about the origin with an arbitrary covariance matrix.

For $1 \leq j \leq k$, we assume that the points of the jth cluster are sampled from a distribution that arises by uniformly scaling all the coordinates of $\mathbf{X}$ by some positive scale factor $a_i \in \mathbb{R}^+$ and translated by some vector $\mathbf{t}_j \in \mathbb{R}^d$. Thus, a point of the jth cluster is generated by a random vector $\mathbf{X}_j = a_i\mathbf{X} + \mathbf{t}_j$. Since the origin is the mean of the base distribution, $\mathbf{t}_j$ is the mean of the jth cluster, which we will call the *distribution center*. Let $Y_j = \|\mathbf{X}_j - \mathbf{t}_j\|$ be the random variable that represents the Euclidean distance from a point of the jth cluster to $\mathbf{t}_j$. Because this is a uniform scaling of the base distribution by a_i and translation by $\mathbf{t}_j$, it is easily verified that $E[Y_j] = a_i E[Y]$ and $E[Y_j^2] = a_i^2 E[Y^2]$.

We make the following additional assumptions about the clusters and the current state of the algorithm's execution:

(1) The clusters are well-separated, that is, the probability that a point belonging to one cluster is closer to the center of another cluster than to its own cluster center is negligible.

(2) The number of points n_j in each cluster is sufficiently large, that is, the law of large numbers can be applied to each cluster. (We do not assume that the clusters have equal numbers of points.)

(3) The algorithm is near convergence, in the sense that the difference between the current location of cluster center $\mathbf{z}_j$ and the actual cluster center $\mathbf{t}_j$ is negligible.

Theorem 6.1 *Subject to Assumptions* (1)–(3) *above, standard* ISOCLUS *and the filtering variant behave identically.*

Proof: As mentioned earlier, the only differences between the two algorithms are in the computations of the individual and average cluster dispersion in Step 5 and their use in determining whether to split a cluster in Step 8. Consider a cluster center j, for $1 \leq j \leq k$. Recall that to establish the equivalence of the two algorithms it suffices to show that

$$\Delta_j > \Delta \text{ (in standard ISOCLUS)} \quad \Longleftrightarrow \quad \Delta'_j > \Delta' \text{ (in filtering ISOCLUS)}.$$

First let us consider the average Euclidean distance of the standard algorithm. Recall that n_j denotes the number of points in a cluster. From the definitions of the cluster distributions and Assumption (3) we have

$$\Delta_j \;=\; \frac{1}{n_j} \sum_{\mathbf{x} \in S_j} \|\mathbf{x} - \mathbf{z}_j\| \;\approx\; \frac{1}{n_j} \sum_{\mathbf{x} \in S_j} \|\mathbf{x} - \mathbf{t}_j\|,$$

where $\approx$ denotes approximate equality (subject to the degree to which Assumption (3) is satisfied).

S_j consists of the points that are closer to $\mathbf{z}_j$ than to any other cluster center. By Assumptions (1) and (3) it follows that the contribution to the dispersion of S_j that arises due to points from other clusters is negligible. From Assumption (2) it follows from the law of large numbers that this quantity, which is just a sample mean of a large number of independent and identically distributed random variables, will be arbitrarily close to the expected value for the cluster distribution. Thus we have

$$\Delta_j \;\approx\; E[\, \|\mathbf{X_j} - \mathbf{t}_j\| \,] \;=\; E[Y_j] \;=\; a_j E[Y].$$

Next, consider the average squared distance of the filtering algorithm. From Assumption (3), the corresponding quantity in this case is

$$\Delta'_j \;=\; \sqrt{\Delta_j^{(2)}} \;=\; \left(\frac{1}{n_j} \sum_{\mathbf{x} \in S_j} \|\mathbf{x} - \mathbf{z}_j\|^2 \right)^{1/2} \;\approx\; \left(\frac{1}{n_j} \sum_{\mathbf{x} \in S_j} \|\mathbf{x} - \mathbf{t}_j\|^2 \right)^{1/2}.$$

As before, from our assumptions we may approximate this sample mean with the expected value for the cluster distribution, from which we obtain

$$\Delta'_j \;\approx\; \sqrt{E[\, \|\mathbf{X}_j - \mathbf{t}_j\|^2 \,]} \;=\; \sqrt{E[Y_j^2]} \;=\; \sqrt{a_j^2 E[Y^2]} \;=\; a_j \sqrt{E[Y^2]}.$$

Now, let us consider the average dispersions computed by the two algorithms. Let $w_j = n_j/n$ denote the fraction of points of S that are in cluster S_j. By the definitions of Δ and Δ' we have

$$\Delta \;=\; \frac{1}{n} \sum_{i=1}^{k} n_i \Delta_i \;=\; \sum_{i=1}^{k} w_i \Delta_i \qquad \text{and} \qquad \Delta' \;=\; \frac{1}{n} \sum_{i=1}^{k} n_i \Delta'_i \;=\; \sum_{i=1}^{k} w_i \Delta'_i.$$

Finally, we combine all of this to obtain the desired conclusion. Observe that the implications are not absolute, but hold in the limit as Assumptions (1)–(3) are satisfied:

$$\begin{tikzcd}
\Delta_j > \Delta \arrow[r, Leftrightarrow] & \Delta_j > \sum_{i=1}^k w_i \Delta_i \arrow[r, Leftrightarrow] & a_j E[Y] > \sum_{i=1}^k w_i a_i E[Y] \arrow[d, Leftrightarrow] \\
 & & a_j > \sum_{i=1}^k w_i a_i \arrow[d, Leftrightarrow] \\
\Delta'_j > \Delta' \arrow[r, Leftrightarrow] & \Delta'_j > \sum_{i=1}^k w_i \Delta'_i \arrow[r, Leftrightarrow] & a_j \sqrt{E[Y^2]} > \sum_{i=1}^k w_i a_i \sqrt{E[Y^2]}
\end{tikzcd}$$

This completes the proof. □

7 Conclusions

We have demonstrated the efficiency of a new implementation of the ISOCLUS algorithm, based on the use of the kd-tree data structure and the filtering algorithm. Our algorithm is a slight modification of the original ISOCLUS algorithm, because it uses squared distances, rather than Euclidean distances as a measure of cluster dispersion in determining whether to split clusters. We have provided both theoretical and experimental justification that the use of squared distances yields essentially the same results. The experiments on synthetic clustered data showed speed-ups in running times ranging from 1.3 to 57, while the experiments on Landsat and MODIS satellite image data showed speed-ups of 4 to 30 and 4 to 22, respectively.

We also presented an approximate version of the algorithm which allows the user to further improve the running time at the expense of lower fidelity in computing the nearest cluster center to each point. We showed that with relatively small distortion errors, significant additional speed-ups can be achieved by this approximate version. The software is freely available, and can be downloaded from `http://www.cs.umd.edu/~mount/Projects/ISODATA`.

One possible direction for future research involves sensitivity to the input parameters. The running times for the standard and hybrid versions increase linearly with the number of points n, the number of centers k, and the dimension d. For the inputs we tested, however, the running time of the filtering version increases sublinearly in n and k, but superlinearly in the dimension d. Thus, the filtering version is most appropriate when n and k are large and the dimension is fairly small.

Acknowledgments

We would like to thank Jeff Morisette of NASA/GSFC and the EOS Validation Core Sites project for providing us with the MODIS data, and In-Joon Chu for his contributions to the geometrical analysis used in approximate filtering. We would also like to thank the anonymous reviewers for their suggestions and insights.

References

[1] P. K. Agarwal, M. Sharir, and E. Welzl. The discrete 2-center problem. *Discrete and Computational Geometry*, 20(3):287–305, 1998.

[2] S. Arora, P. Raghavan, and S. Rao. Approximation schemes for Euclidean k-median and related problems. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pages 106–113, Dallas, TX, May 1998.

[3] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. *Journal of the ACM*, 45:891–923, 1998.

[4] V. Arya, N. Garg, R. Khandekar, V. Pandit, A. Meyerson, and K. Munagala. Local search heuristics for k-median and facility location problems. *SIAM Journal of Computing*, 33(3):544–562, 2004.

[5] G. H. Ball and D. J. Hall. Some fundamental concepts and synthesis procedures for pattern recognition preprocessors. In *Proceedings of the International Conference on Microwaves, Circuit Theory, and Information Theory*, Tokyo, Japan, September 1964.

[6] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[7] L. Bottou and Y. Bengio. Convergence properties of the k-means algorithms. In G. Tesauro and D. Touretzky, editors, *Advances in Neural Information Processing Systems 7*, pages 585–592. MIT Press, Cambridge, MA, 1995.

[8] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *Proceedings of the Fortieth Annual Symposium on Foundations of Computer Science*, pages 378–388, New York, NY, October 1999.

[9] T. Feder and D. H. Greene. Optimal algorithms for approximate clustering. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 434–444, 1988.

[10] W. Feller. *An Introduction to Probability Theory and Its Applications.* John Wiley & Sons, New York, NY, third edition, 1968.

[11] E. Forgey. Cluster analysis of multivariate data: Efficiency vs. interpretability of classification. *Biometrics*, 21:768, 1965.

[12] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3:209–226, 1977.

[13] K. Fukunaga. *Introduction to Statistical Pattern Recognition.* Morgan Kaufman, San Diego, CA, 1990.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, New York, NY, 1979.

[15] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression.* Kluwer Academic, Boston, MA, 1992.

[16] T. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[17] S. Har-Peled. Clustering motion. In *Proceedings of the Forty Second IEEE Symposium on Foundations of Computer Science*, pages 84–93, Washington, DC, October 2001.

[18] S. Har-Peled. Clustering motion. *Discrete and Computational Geometry*, 31(4):545–565, 2004.

[19] S. Har-Peled and A. Kushal. Smaller coresets for k-median and k-means clustering. In *Proceedings of the Twenty First Annual ACM Symposium on Computational Geometry*, pages 126–134, Pisa, Italy, June 2005.

[20] S. Har-Peled and S. Mazumdar. Coresets for k-means and k-median clustering and their applications. In *Proceedings of the Thirty Sixth Annual ACM Symposium on Theory of Computing*, pages 291–300, Chicago, IL, June 2004.

[21] S. Har-Peled and B. Sadri. How fast is the k-means method? *Algorithmica*, 41(3):185–202, January 2005.

[22] D. S. Hochbaum. *Approximation Algorithms for NP-hard Problems.* PWS Publishing Company, Boston, MA, 1997.

[23] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[24] K. Jain and V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 2–13, New York, NY, October 1999.

[25] J. R. Jensen. *Introductory Digital Image Processing: A Remote Sensing Perspective.* Prentice Hall, Upper Saddle River, NJ, second edition, 1996.

[26] T. Kanungo, D. M. Mount, N. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. A local search approximation algorithm for k-means clustering. *Computational Geometry: Theory and Applications*, 28:89–112, 2004.

[27] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24:881–892, 2002.

[28] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis.* John Wiley & Sons, New York, NY, 1990.

[29] T. Kohonen. *Self-Organization and Associative Memory.* Springer-Verlag, New York, NY, third edition, 1989.

[30] S. Kolliopoulos and S. Rao. A nearly linear-time approximation scheme for the Euclidean k-median problem. In J. Nesetril, editor, *Proceedings of the Seventh Annual European Symposium on Algorithms*, volume 1643 of *Lecture Notes in Computer Science*, pages 362–371. Springer-Verlag, July 1999.

[31] A. Kumar, Y. Sabharwal, and S. Sen. A simple linear time $(1 + \epsilon)$-approximation algorithm for k-means clustering in any dimensions. In *Proceedings of the Forty Fifth Annual IEEE Symposium on Foundations of Computer Science*, pages 454 – 462, Rome, Italy, October 2004.

[32] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

[33] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–296, Berkeley, CA, 1967.

[34] O. L. Mangasarian. Mathematical programming in data mining. *Data Mining and Knowledge Discovery*, 1:183–201, 1997.

[35] J. Matoušek. On approximate geometric k-clustering. *Discrete and Computational Geometry*, 24:61–84, 2000.

[36] R. T. Ng and J. Han. CLARANS: A method for clustering objects for spatial data mining. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1003–1016, 2002.

[37] PCI Geomatics Corp. ISOCLUS–Isodata clustering program. `http://www.pcigeomatics.com/cgi-bin/pcihlp/ISOCLUS`.

[38] D. Pollard. A central limit theorem for k-means clustering. *Annals of Probability*, 10:919–926, 1982.

[39] J.A. Richards and X. Jia. *Remote Sensing Digital Image Analysis.* Springer, Berlin, 1999.

[40] S. Z. Selim and M. A. Ismail. K-means-type algorithms: A generalized convergence theorem and characterization of local optimality. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:81–87, 1984.

[41] M. Sharir. A near-linear algorithm for the planar 2-center problem. *Discrete and Computational Geometry*, 18:125–134, 1997.

[42] J. T. Tou and R. C. Gonzalez. *Pattern Recognition Principles.* Addison-Wesley, London, 1974.